by
Ray Duncan

# Power Programming

## A Look at the Differences Between C and C++

**■ If you're a veteran C programmer, you may have found C++'s odd notation disconcerting. Luckily, using C++ isn't an all-or-nothing proposition—you can make the move incrementally and (for the most part) painlessly.**

In the spring of 1988, I participated in a panel called "The Language Wars" at the West Coast Computer Faire in San Francisco. Two of the other panelists were Greg Lobdell, the programming-language evangelist for Microsoft, and David Intersimone, Borland's evangelist. These two worthy fellows presented ideas that were very similar: Both expressed a heartfelt commitment to object-oriented programming tools, both paid appropriate obeisance to the imminent triumph of graphical user interfaces, and both predicted that integrated development environments, debuggers, profilers, and project management facilities would become the new competitive battlegrounds as compiler technology matured.

We are now in a position to put these predictions to the test of time and to judge the promoters' words by their companies' actions. As I write this, Borland has shipped no fewer than five different object-oriented programming packages (Turbo Pascal 5.5, Turbo Pascal 6.0, Turbo Pascal for *Windows* (TPW), Turbo C++ 1.0, and Borland C++ 2.0), as well as a novel object-oriented user interface builder (ObjectVision), and has produced the only GUI-based mainstream language for *Windows* on the market to date, TPW. The Borland integrated development environments have evolved steadily, and in their current incarnations they rival high-end programming editors like *Brief* for features and customizability. The auxiliary Borland tools, like TASM, TLINK, Turbo Profiler, and Turbo Debugger, are speedy, well-designed, and a pleasure to use, and the Turbo Debugger for *Windows* is a marvel. The Borland compilers, linker, and debugger can take advantage of 80286 and 80386 features, but they'll still run (and run very well) on 8086 machines with 640K of memory. Finally, the Borland VROOM overlay manager makes it feasible to write large, complex applications that will execute efficiently on even the oldest, slowest PCs.

Microsoft, on the other hand, struggled and labored to bring forth MSC 6.0, and ultimately produced a product that was an elephant in terms of its demand for machine resources and a mouse in terms of performance. The Microsoft integrated development environment, called Programmer's Workbench, is so big, slow, and clumsy that it is virtually unusable under DOS, and the third version of Microsoft CodeView is still nowhere near as slick and intuitive as the very first version of Turbo Debugger. The new MAKE program (NMAKE) is even weirder than the old one. Microsoft's compiler documentation, formerly excellent, has taken a distinct turn for the worse; you can only get the complete runtime library manual by purchasing a separate book not included in the retail MSC package. As for those Microsoft object-oriented programming languages—C-Sharp and Visual Basic—and the GUI-based development platforms we've been hearing about for so long? They still haven't materialized, six years after the appearance of *Microsoft Windows* and three years after the release of OS/2 Presentation Manager.

When I began to dabble in *Windows* development again after spending several years with OS/2, I was appalled to see how little Microsoft had done in the meantime to make the lives of *Windows* programmers easier. It's certainly a tribute to the talents of the Microsoft Applications Group that they've been able to bring superb products like *Word for Windows* and *Excel* 3.0 to market while being forced to rely on such primitive tools. But the rest of us are free to choose among language vendors, and it took only a few days using Borland C++ 2.0 to convince me to abandon the Microsoft compiler and *Windows* SDK completely. Aside from the overall high quality and performance of BC++, its support for precompiled header files, its Project Manager, which eliminates the need for arcane MAKE and NMAKE files, its multiwindow editor with multilevel undo/redo, and the single-screen Turbo Debugger for *Windows* were more than enough incentive for me to switch and never look back. Borland C++ is not yet perfect—Borland needs to pay some serious attention to code optimization, needs to move BC++ to a true GUI-based implementation like TPW, and needs to add source-code control facilities to the integrated development environment—but it's clear that Borland has taken a commanding lead in programming-tools technology as a whole.

### VIEWING C++ AT ARM'S LENGTH

If you're a veteran C programmer raised on the K&R (Kernighan and Ritchie) flavor

of the language and your curiosity led you to peek at C++ books, you were probably aghast at the bizarre notation and terminology you found. That's how I reacted the first few times I tried to learn about C++. If ordinary C code borders on the cryptic, C++ code can be downright grotesque, and it uses several traditional C tokens and symbols (such as : and << and >>) in unfamiliar ways. Furthermore, Bjarne Stroustrup, the inventor of C++, invented new and unobvious names for the object-oriented elements of his language, rather than adopting or adapting terms used in predecessor OOP languages like Smalltalk. The effect is that of a bad dream where everyone is using words that sound familiar but don't make sense.



Fortunately, using C++ is not an all-or-nothing proposition. Borland C++ actually consists of two compilers in one development environment: an ANSI C compiler and a C++ compiler that's compliant with the AT&T C++, Version 2.0, specification. The project manager determines which compiler to run against a given source code module by examining the module's extension: .C or .CPP. This allows you to make the move to C++ incrementally and (for the most part) painlessly. You can simply create Borland C++ project files from your old Make files, make any changes necessary to get the program to compile and link properly using the ANSI C compiler, and then migrate the source modules individually to C++ by editing them appropriately and changing their filename extension. Further, since C++ syntax is largely a superset of C, this approach lets you take immediate advantage of some highly useful C++ features without drastically restructuring your programs.

The biggest hurdle you'll have to surmount, if you haven't updated your original C program for ANSI C, is to recast every one of your function declarations in the "new style" and add function prototypes to your application's header files. In K&R's original definition of C, prototypes were unknown, and functions were declared in the following form:

```
myfunc(x, y)
int x;
int y;
{
   ...
}
```

Functions were assumed to return an integer unless otherwise specified. This could lead to problems when modules were compiled separately, because a reference in one module to a function that was defined in another module might assume (for instance) that the function returned an integer, when it actually was declared as returning a double. Neither the compiler nor the linker would report an error in such cases, but the program would behave unpredictably or crash. The possible solutions were not elegant: Either all the functions that referenced a noninteger function had to be defined after that function in the same module, or each calling function had to declare the type of each external noninteger function that it used along with its local data declarations.

As the language evolved, most C

compilers were enhanced to support function prototypes, which were declarations of a function's value type (separate from the definition of the function) that could be placed in a header file and #included in all modules of a program. The concept of a *void* type was also introduced to allow the compiler to flag the misuse of functions that didn't actually return anything. Such a post-K&R but pre-ANSI function prototype and declaration would look like this:

```
/* function prototype in header
   file */
void myfunc();
.
.
.
/* declaration of function */
void myfunc(x, y)
int x;
int y;
{
     ...
}
```

This was a major improvement on the original K&R C, but something was still missing. The prototypes didn't include declarations of the types of the function's arguments, so the compiler still couldn't cross-check the parameters being passed to a function by its callers in other modules against the actual parameter types understood by the function. Stroustrup answered this need in C++ by introducing complete, formal prototypes, and incidentally changing the declarations of functions so that the types and names of its parameters were found together. For example,

```
/* function prototype in header
   file */
void myfunc(int, int);
.
.
.
/* declaration of function */
void myfunc(int x, int y)
{
     ...
}
```

This style makes it possible for the compiler to perform complete parameter and return value type-checking on every invocation of a function. The ANSI C standardization committee, which was meeting during the period that C++ was becoming stabilized, thought this was such a neat idea that it subsumed Stroustrup's style of function prototyping and declaration wholesale into ANSI C—incidentally transforming C from a weakly typed, anything-goes language into a strongly typed language in the process, and making it much more practical to create optimizing C compilers.

I should note that while ANSI C encourages use of the new style of prototype and function declaration, but continues to support the old style with minor limitations, C++ demands complete function prototypes. Since the explicit prototypes are easiest to generate and keep in sync if you adopt the new style of function declaration, you might as well make this conversion globally throughout your application as you bring it into compliance with ANSI C. This should make migration of the application to the C++ compiler less painful. Once you've taken this step, however, C++ offers a bonus: the ability to specify default argument values. For example, imagine a graphics drawing function—circle()—called with the coordinates of the origin and a radius:

**An interesting aspect of C++ is that it lets you declare variables close to the point where they are used.**

```
void circle (int x, int y,
 int radius=100)
{
     ...
}
```

This C++ function declaration permits the function circle() to be called with either two arguments or three. If the caller doesn't supply a radius, it defaults to 100. On the other hand, if the caller fails to supply the **x** or **y** arguments, the compiler will generate an error message.

Another interesting, non-object-oriented aspect of C++ is that it allows you to declare variables and structures close to the point where they are used, in contrast to C's requirement that all data declarations be grouped at the beginning of a function. For example, in ANSI C you have to write

```
void circle (int x, int y,
 int radius)
{
   int i;  /* a temporary loop
    index */
   int j;  /* another temporary
    index */
   .
   .   /* executable code here */
   .
   for(i = 0; i < radius; i++)
   {   for(j = 0; j < radius; j++)
       {
          ...
       }
   }
}
```

while in C++ you're allowed to write

```
void circle (int x, int y,
 int radius)
{
   .
   .   /* executable code here */
   .
   int i, j;
   for(i = 0; i < radius; i++)
   {
       for(j = 0; j < radius; j++)
       {
          ...
       }
   }
}
```

In many cases, you can even declare a variable in the context of its first usage, such as

```
for(int i=0; i < radius; i++)
```

But this style can also get you into trouble, so it's probably best avoided for now; the restrictions on it aren't obvious until you've learned more about C++'s object allocation and initialization.

Yet another elegant feature of C++ is its improved interface for dynamic memory management. While C++ still supports the traditional C functions malloc() and free(), it also offers two superior func-

tions: new() and delete(). These functions know about the sizes of objects automatically, making it unnecessary for you to determine those sizes programmatically, and incidentally improving the readability of the source code and eliminating many opportunities for subtle bugs. For example, in C, you might allocate a pointer to an array of double-precision (64-bit) floating-point numbers and then allocate memory for the array itself, with code something like this:

```
double *darray;
.
.
.
darray = malloc(50 *
 sizeof(double));
```

In C++, you'd allocate the pointer and the memory in the same place with code like this:

```
double *darray = new double[50];
```

Last but not least, we should mention the C++ single-line comment, whose initiator is a double forward-slash. As you know, traditional C comments are delimited by a /* token at the beginning and a */ token at the end, for example:

```
int i;  /* this is a comment */
```

How many times have you been booby-trapped by accidentally deleting, damaging, or nesting one of these comment delimiters in a C source file? Often C compilers give you no useful information at all about such an accident, instead producing either a cascade of error messages, or no error messages at all except for "unexpected end of file." In C++, you can write short comments like this:

```
int i;  // this is a comment
```

The end-of-line serves as the comment terminator. This idea of Stroustrup's has proven so convenient that most ANSI C compilers support it even though it's not part of the ANSI C standard.

### OBJECT-ORIENTED PROPERTIES OF C++

Now that we've seen some of the characteristics of C++ that we can exploit without abandoning a C-language point of view, let's turn to C++'s support for object-oriented programming. Recall that there are three features of a language that mark it as truly object-oriented: encapsulation, polymorphism, and inheritance.

*Encapsulation* is a fancy term for information hiding. In a classic object-oriented programming language, procedures (methods) and their variables and constants are hidden inside entities called *objects*, which communicate by means of messages. Each object is an instantiation of an object type or class; the members of a class have different names but symmetric capabilities. The benefit of encapsulation is that a particular object class's implementation can be changed and improved without causing harmful side effects elsewhere.

*Polymorphism* refers to an object's capability to select the correct internal procedure (method) based on the type of data received in a message. For example, a "Print" object might be sent a message containing a binary integer, a binary floating-point number, or an ASCII string; in an object-oriented programming language, you have a right to expect the object to take the appropriate action (or at least fail gracefully) even if the contents of the message aren't known at the time you write the program.

*Inheritance* refers to a language's mechanism for deriving a new class of objects from an existing class (subclassing). The new class need only contain the actual code and data for new or changed methods; the code and data for methods that are inherited unchanged from the parent class remain in the parent, and messages not handled by a particular class's unique methods are automatically routed to its parent. Some languages support multiple inheritance—the possibility that a class can inherit methods from two or more completely unrelated parent classes. This is a hotly controversial issue among OOP mavens, and therefore it's one we'll ignore for now.

C++ meets all three of these classic criteria of object-orientedness, but once you really start to take advantage of them in your programs, you'll find you're definitely not in Kansas anymore. Although C++ has just a few more keywords and operators than C, Stroustrup gets a lot of mileage out of them, and while C++ programs can superficially look very much like C programs, they are put together in strange and wonderful ways.

Consider a very simple test-bed program that prompts a user to enter a number and a string, then displays those same items back. In ANSI C, the code would look something like this:

```
int i;
char temp[80];
char buff[80];

printf("Enter a number: ");
gets(temp);
i = atoi(temp);

printf("Enter a string: ");
gets(buff);

printf("Your number is: %d\n", i);
printf("Your string is: %s\n",
 buff);
```

In C++, the code for the same job would look very different:

```
int i;
char buff[80];

cout << "Enter a number: ";
cin  >> i;
cout << "Enter a string: ";
cin  >> buff;

cout << "Your number is: "
 << i    << "\n";
cout << "Your string is: "
 << buff << "\n";
```

This little chunk of code actually illustrates C++'s object-oriented nature very well. **cin** is an object that obtains user input, **cout** is an object that performs formatted output. When you send or get a message to or from **cin** or **cout**, they look at the type of data being requested or sent and take the appropriate actions (polymorphism); the details of how each works are (or at least can be) completely transparent to the caller (encapsulation), and both are instances of subclasses of a more basic class called **iostream** (inheritance).

In the next issue, we'll explore C++'s object-oriented features further.

### THE IN-BOX

Please send your questions, comments, and suggestions to me at any of the following e-mail addresses:

PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan